A TECHNIQUE FOR DOING LAZY EVALUATION IN LOGIC

SANJAI NARAIN

▷ We develop a natural technique for defining functions in logic, i.e. PROLOG, which directly yields lazy evaluation. Its use does not require any change to the PROLOG interpreter. Function definitions run as PROLOG programs and so run very efficiently. It is possible to combine lazy evaluation with nondeterminism and simulate coroutining. It is also possible to handle infinite data structures and implement networks of communicating processes. We analyze this technique and develop a precise definition of lazy evaluation for lists. For further efficiency we show how to preprocess programs and ensure, using logical variables, that values of expressions once generated are remembered for future access. Finally, we show how to translate programs in a simple functional language into programs using this technique. ◁

1. INTRODUCTION

Lazy evaluation is a scheme for evaluation of purely functional expressions which ensures that an expression is evaluated only when there is demand for its value.[1] In particular, the scheme can determine the value of an expression in finite time, even if that expression contains subexpressions representing infinite structures.

Several implementations of lazy evaluation for functional languages have been proposed [6, 11, 24]. Delayed evaluation and the closely related idea of coroutining have also been proposed and implemented for logic based systems [4, 8, 10, 14, 22, 23]. However, all these approaches have suggested keeping the functional or logic

Address correspondence to Sanjai Narain, Information Science Department, Rand Corporation, Santa Monica, CA 90406.

Received

[1] A secondary, though useful, property usually associated with lazy evaluation is that the value of an expression, once determined, is retained for future access.

THE JOURNAL OF LOGIC PROGRAMMING

52 Vanderbilt Ave., New York, NY 10017 0743-1066/86/$03.50

language fixed but modifying the usual (but highly efficient) methods of interpreting programs in these languages. This is a major reason why, with the possible exception of Turner's [24], the modified interpreters are not efficient enough for practical programming.

We describe a technique for defining functions in PROLOG that directly yields lazy evaluation. Rather than modifying the interpreter, we modify the programming style *but continue to keep it natural.* Programs run as PROLOG programs and so take full advantage of the efficient PROLOG implementations available today. We also retain the benefit of PROLOG's powerful features, principally, unification pattern matching, nondeterminism, partially instantiated structures, and a deductive machinery.

In the following sections we (1) informally develop the technique and give several examples of its applications, (2) formalize ideas behind it and give a precise definition of lazy evaluation for lists, (3) introduce a device which makes use of the nature of logical variables to implement "updating" of expression values, (4) show how to translate programs in a simple functional language into programs using this technique, and (5) outline connections of our ideas with related work.

2. DEVELOPING THE TECHNIQUE

Given a finite expression, the problem is to determine its value. One way is to repeatedly reduce the expression using a set of reduction rules till it is no longer possible to reduce it further. The final expression then represents the value of the original expression. A set of reduction rules together with a strategy for applying them may be called a *reduction scheme.* A reduction scheme must satisfy two conditions. First, it must preserve value through every reduction, i.e., the value of an expression before an application of the reduction rules must be the same as the value of the expression obtained after the application. Second, the scheme must compute the value whenever it is computable, i.e., the value must be found merely by repeated use of the strategy for applying rules. For the lambda calculus, alpha and beta reduction rules combined with the normal-order reduction strategy represent one scheme satisfying these two conditions. As Turner [24] has pointed out, this scheme directly yields lazy evaluation. It is well known that an applicative order reduction strategy, such as that of conventional LISP, leads to termination less often than does the normal-order one.

We present a technique for defining functions in PROLOG which leads to a reduction scheme directly yielding lazy evaluation. The scheme, however, is characterized by conditions somewhat different from those above. First, each reduction must preserve the value of an expression. Second, each reduction must transform a *finite expression* into another *finite expression* and result in *useful simplification.* In the case of expressions denoting lists, useful simplification means the following: If an expression E denotes a list L, then E is reduced either to the empty list [ ], or to the form $[U|V]$, where U is in *irreducible*, i.e. *printable*, form and denotes the head of L, and V is a finite expression denoting the tail of L.

Infinite structures may be easily dealt with using such a technique provided finite representations of these structures are always reduced into other finite representations and result in useful simplifications.

We now informally develop this technique using some familiar examples, leaving a formal description to Section 6. For each n-ary function f we will write reduction rules satisfying the above two conditions. The rules will specify how to reduce the expression f($X1$, $X2$,..., Xn), where each Xi represents a formal parameter of f. As may be seen from the examples below, the appropriate reduction rules for a function f can be written down almost directly from the usual functional definition of f. (In Section 8 we show how to do this automatically). For example, in a LISP-like language, the definition of the function intfrom for computing the list of natural numbers starting at N is

$$\text{intform}(N) = \text{cons}(N, \text{intfrom}(N+1)).$$

The appropriate reduction rule in PROLOG is

$$\text{reduce}(\text{intfrom}(N), [N|\text{intfrom}(N1)]) :- N1 \text{ is } N+1.$$

One application of this reduction rule to the finite expression intfrom(N) reduces it to another finite expression [N|intfrom($N1$)], where $N1$ is $N+1$. Clearly, the value of [N|intfrom($N1$)] is the same as the value of intfrom(N), the head of the list denoted by intfrom(N) is computed in irreducible form as N, and a finite representation of its tail is computed as intfrom($N1$). If we now type the query

$$\text{reduce}(\text{intfrom}(1), Z).$$

the PROLOG interpreter will half in finite time and bind Z to [1|intfrom(2)]. It will be noted that application of a reduction rule is accomplished by a call to reduce. To compute, for example, the third element of the list of intfrom(1) we may type the query

$$\text{reduce}(\text{intfrom}(1), [U|V]), \text{reduce}(V, [A|B]), \text{reduce}(B, [P|Q]).$$

which will also succeed with the bindings $U = 1$, $V =$ intfrom(2), $A = 2$, $B =$ intfrom(3), $P = 3$, $Q =$ intfrom(4). The third element is then the value of P. Thus, by a suitable sequence of calls to reduce, we can compute, in finite time, any element of the list intfrom(1), even though the list itself is infinite. This is the kind of behavior we expect from a lazy evaluation scheme.

For the next example we consider the definition of the append function in a LISP-like language:

$$\text{append}(X, Y) = \text{if null}(X) \text{ then } Y \text{ else cons}(\text{car}(X), \text{append}(\text{cdr}(X), Y))$$

The corresponding reduction rules are

$$\text{reduce}(\text{append}(X, Y), Z) :- \text{reduce}(X, [\,]), \text{reduce}(Y, Z).$$

$$\text{reduce}(\text{append}(X, Y), [FX|\text{append}(RX, Y)]) :- \text{reduce}(X, [FX|RX]).$$

We can easily verify (as shown in Section 6) that if reduce(append(X, Y), E) then the value of append(X, Y) is the value of E, and that E is a useful simplification of append(X, Y). To show how this program works on a simple example we must first define PROLOG lists (i.e. the functions | and [ ]):

$$\text{reduce}([U|V], [U|V]).$$

$$\text{reduce}([\,], [\,]).$$

If we now execute the query reduce(append([1, 2], [3, 4]), [U|V]), it succeeds with the

bindings $U = 1$ and $V =$ append([2],[3,4]). Thus the list append([1,2],[3,4]) is only partially evaluated, as is our intention. To fully evaluate a list, we may define

```
make_list(L,[]) :- reduce(L,[]).
make_list(L,[FL|V]) :- reduce(L,[FL|RL]), make_list(RL,V).
```

If we now execute make_list(append([1,2],[3,4]), Z), we get $Z = [1,2,3,4]$. Moreover, the number of reduction steps (three) necessary to compute the entire list is the same as that necessary with the functional definition of append, or with its usual PROLOG definition.

3. SIEVE OF ERATOSTHENES

We now give an implementation, using this technique, of the sieve-of-Eratosthenes algorithm [15] for computing a list of all prime numbers. The reduce rules are given directly and may be seen to correspond naturally to the usual functional definition:

```
reduce(primes, Z) :- reduce(sieve(intfrom(2)), Z).
reduce(intfrom(N), [N|intfrom(N1)]) :- N1 is N + 1.
reduce(sieve(L), [FL|sieve(filter(FL, RL))]) :- reduce(L,[FL|RL]).
reduce(filter(A, L),[]) :- reduce(L,[]).
reduce(filter(A, L),[FL|filter(A, RL)]) :-
    reduce(L,[FL|RL]), not(multiple(FL, A)).
reduce(filter(A, L), Z) :-
    reduce(L,[FL|RL]), multiple(FL, A), reduce(filter(A, RL), Z).
```

If we now define

```
print_list(L) :- reduce(L,[FL|RL]), write(FL), write(' '), print_list(RL).
```

and then execute print_list(primes), we get

2 3 5 7 11 13 ...

till the system runs out of stack space. We thus have a natural and efficient implementation of this algorithm in conventional PROLOG.

4. COMBINING LAZY EVALUATION WITH NONDETERMINISM

One of the most powerful features of PROLOG is a facility for writing nondeterministic programs. An object may be defined as a set of constraints that it must satisfy. Then, all objects that satisfy those constraints are automatically generated. Of course, a nondeterministically defined object may itself be defined in terms of nondeterministically defined objects. This makes for highly concise implementations of generate and test strategies.

It is often desirable to interleave the generation of an object with testing whether it satisfies its constraints. If at any stage the partially generated object does not satisfy its constraints it is not necessary to generate it further. A facility for coroutining makes it possible to support such interleaving, e.g. in IC-PROLOG [4]. Lazy evaluation, by its very definition, achieves much the same effect.

Of course, nondeterminism and lazy evaluation are two distinct ideas. We now give an example, a program for computing permutations of a list, and show how these two ideas may easily be combined using our technique. We then give a program for solving the eight-queens problem which behaves as efficiently as the improved program in [7] for solving the same problem.

The usual nondeterministic PROLOG program for computing permutations of a list is

```
perm([ ],[ ]).
perm(U,[A|V]) :- remove(U, A, B), perm(B, V).

remove([U|V], U, V).
remove([U|V], A, [U|B]) :- remove(V, A, B).
```

The appropriate reduce rules are

```
reduce(perm(L),[ ]) :- reduce(L,[ ]).
reduce(perm(L),[A|perm(B)]) :- reduce(remove(L),[A|B]).

reduce(remove(L),[U|V]) :- reduce(L,[U|V]).
reduce(remove(L),[A,U|B]) :- reduce(L,[U|V]), reduce(remove(V),[A|B]).
```

If we now execute reduce(perm([1,2,3]),[A|B]), we get the three answers: $A = 1$, B = perm([2,3]); $A = 2$, B = perm([1,3]); $A = 3$, B = perm([1,2]). We see that the form perm([1,2,3]) reduces only partially, but in three different ways, and thus both lazily and nondeterministically.

The rest of the program for the eight queens problem is now given. As is well known, the problem consists of placing eight queens on a chessboard in such a way that no two queens can attack one another. Since under this condition, no two queens can be in the same column, the problem reduces to finding a suitable permutation of the list [1,2,3,4,5,6,7,8], where the nth element specifies the row number of the queen in the nth column:

```
reduce(queens(L), Z) :- reduce(safe(perm(L)), Z).
reduce(safe(L),[ ]) :- reduce(L,[ ]).
reduce(safe(L),[Q|safe(nodiagonal(Q,List,1))]) :- reduce(L,[Q|List]).

reduce(nodiagonal(_, X, _),[ ]) :- reduce(X,[ ]).
reduce(nodiagonal(Q, X, N),[Q1|nodiagonal(Q, L, N1)]) :-
    reduce(X,[Q1|L]), N1 is N + 1, noattack(Q, Q1, N).

noattack(Q1, Q2, N) :- Q1 > Q2, Diff is Q1-Q2, Diff =\= N.
noattack(Q1, Q2, N) :- Q1 < Q2, Diff is Q2-Q1, Diff =\= N.
```

To obtain solutions one may type make_list(queens([1,2,3,4,5,6,7,8]), Z). The program proceeds by placing one queen at a time and checking if it conflicts with any queen already placed.

5. EXTENSION TO TREES

We now give a PROLOG implementation of an algorithm for comparing the fringe of two binary trees. A lazy implementation has the advantage that generation of fringes stops as soon as an element in one fringe is discovered to be unequal to the corresponding element in the other fringe. The work of (redundantly) generating the remainder of fringes is thus saved.

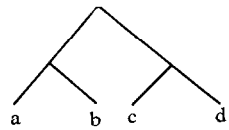
The representation of binary trees is based upon that in [2]. A binary tree is either a fringe node, or it consists of a left subtree and a right subtree which are also binary. An appropriate definition of useful simplification of expressions denoting binary trees is: If E denotes a binary tree, then either E is reduced to tip(X) where X is a label, or E is reduced to $t(L, R)$ where L and R are finite representations of the left and right subtrees respectively of the value of E.

As we defined PROLOG lists above, we can define the functions tip and t as follows:

reduce($t(X, Y), t(X, Y)$).

reduce(tip(X), tip(X)).

The binary tree



may be represented by $t(t(\text{tip}(a), \text{tip}(b)), t(\text{tip}(c), \text{tip}(d)))$.

The function flatten that computes the fringe of a binary tree X is defined as:

reduce(flatten(X), Z) :- reduce(X, $t(LX, RX)$),
    reduce(append(flatten(LX), flatten(RX)), Z).

reduce(flatten(X), [U]) :- reduce(X, tip(U)).

We may now define

eqlist(X, Y) :- reduce(X, [ ]), reduce(Y, [ ]).

eqlist(X, Y) :- reduce(X, [$A|RX$]), reduce(Y, [$A|RY$]), eqlist(RX, RY).

To compare fringes of two binary trees X and Y we execute the query

eqlist(flatten(X), flatten(Y)).

The computation proceeds by generating one fringe node at a time for each tree, comparing them, and if the comparison is successful, proceeding to generate and compare the rest of the fringes.

We have shown in this example how the technique may be used for dealing with functions that compute trees. We may define, in a similar fashion, any function that computes a data structure having a fixed number of fields and a fixed number of primitive versions (e.g. the empty list in the case of lists). The computation of these functions would proceed efficiently but also lazily.

6. AN ANALYSIS OF THE TECHNIQUE

We now formalize ideas presented in previous sections and attempt to understand how they have the potential to yield lazy evaluation. We point out that just the use

of reduce rules may not in itself lead to lazy evaluation. For example, if we rewrite the definition of intfrom as

reduce(intfrom(N), [$N|Z$]) :- $N1$ is $N+1$, reduce(intfrom($N1$), Z).

the query reduce(intfrom(2), [$U|V$]) will never terminate, even though U exists.

We define a first-order language called **Lists1** in which all well-formed formulas (also called terms) will be ground. Further, the value of each term will be a list. We define reduction rules for reducing terms in **Lists1** to other terms in **Lists1**. We then define conditions that must be satisfied by all terms in **Lists1** in the context of these rules. These conditions, if satisfied, will ensure that reductions of terms in **Lists1** will be carried out lazily—according to a definition of lazy evaluation for lists, which we then state. Finally, we show that all terms in **Lists1** do indeed satisfy the above conditions. It should be possible to extend such a treatment to other data types.

In the following, by PROLOG, *we mean any implementation of SLD-resolution* [13]. The metalanguage for **Lists1** is a combination of PROLOG and English. The reduction rules are, however, written in PROLOG. The definition of **Lists1** is as follows:

Alphabet.

An enumerably infinite list of irreducible elements. These elements are like the quoted expressions in Lisp.

The 0-ary function symbol [ ]

The 1-ary function symbol intfrom

The 2-ary function symbols |, append, filter

The usual punctuation marks

Formation rules.

[ ] is a term.

If X is an irreducible element and Y is a term then [$X|Y$] is a term.

If X is an integer then intfrom(X) is a term.

If X and Y are terms then append(X, Y) is a term.

If A is an integer and X is a term then filter(A, X) is a term.

Note that no variables occur in terms of **Lists1**.

Values of terms.

The value of an irreducible element is an item which can occur in a list. No two distinct irreducible elements can have the same value.

The value of [ ] is the empty list.

The value of [$X|Y$] is the list whose head is the value of X and whose tail is the value of Y.

The value of intfrom(X) is the infinite list of natural numbers starting at X.

The value of append(X, Y) is the list obtained by concatenating the value of X with the value of Y.

If A and X have values A^* and X^*, then the value of filter (A, X) is the list obtained by deleting all multiples of A^* from X^*. The elements of this list occur in the same order as they occur in X^*. However, if X^* is an infinite list, then there must be an infinite number of elements of X^* which are not multiples of A^*; otherwise the value of filter(A, X) is not defined. For example, the value of filter(1, intfrom(2)) is not defined.

Reduction rules. These are written in PROLOG. The predication reduce(X, Y) means that the term X can be reduced to the term Y:

```
reduce([],[]).

reduce([U|V],[U|V]).

reduce(intfrom(N),[N|intfrom(N1)]) :- N1 is N + 1.

reduce(append(X, Y), Z) :- reduce(X,[]), reduce(Y, Z).

reduce(append(X, Y),[FX|append(RX, Y)]) :- reduce(X,[FX|RX]).

reduce(filter(A, X),[]) :- reduce(X,[]).

reduce(filter(A, X),[FX|filter(A, RX)]) :-
    reduce(X,[FX|RX]), not(multiple(FX, A)).

reduce(filter(A, X), Z) :-
    reduce(X,[FX|RX]), multiple(FX, A), reduce(filter(A, RX), Z).
```

We now define two conditions that must be satisfied by every term in **Lists1** in the context of these reduction rules. Informally, each term must be reduced to another term of equal value and must be "usefully" simplified. These conditions stated formally are:

Soundness condition. $S(E)$ = For all Z belonging to **Lists1**, if reduce(E, Z), then if the value of E exists, then the value of Z exists and is the same as the value of E.

Completeness condition. $C(E)$ = conjunction of the following:

(a) If the value of E is the empty list then reduce(E,[]).

(b) If the value of E is a nonempty list, then there exists an irreducible element U and a term V such that reduce(E,[$U|V$]).

Note that U must always be an irreducible element, so it does not have to be "coerced" further. This requirement is important. It ensures that after one reduction step, the list is simplified in such a way that it no longer remains necessary to consider the head of the list again for simplification.

If the above two conditions are satisfied by every term in **Lists1**, then given a term E whose value is a list, we can compute each element of this list exclusively by calls to the predicate reduce. In particular, we can use the following PROLOG program to print this list out:

```
print_list(L) :- reduce(L,[]), write(' '), write(nil).

print_list(L) :- reduce(L,[U|V]), write(U), write(' '), print_list(V).
```

The above program represents a reduction strategy: To reduce the term $f(X1, X2, \ldots, Xn)$ completely, apply to the whole term the rule corresponding to f. If the term reduces to [ ], then print nil and halt; otherwise the term reduces to the form $[U|V]$, in which case print U and reduce V completely. We may now define:

Definition. Given a language L in which terms denote lists, a reduction scheme R for reducing terms in L is lazy iff: for any term E, if the value of E is the empty list, then R prints out nil in fiite time; otherwise R prints out the representation, in L, of every member of the value of E in finite time.

In other words, we are proposing indentifying the notion of lazy evaluation with that of completeness, which is, roughly, that if an answer exists then it is computable. In the present context the precise notion of completeness is as expressed by the condition C above or in the above definition of laziness. The major objection to applicative order reduction is that it sometimes fails to compute the normal form of an expression even when it exists [e.g. of the expression $KI\Omega$, where K is $\lambda x.\lambda y.x$, I is $\lambda x.x$, and Ω is $(\lambda x.xx)\lambda u.uu$]. This difficulty is overcome in the presence of a complete reduction procedure, such as normal-order reduction.

If we are able to write down reduce rules such that in their context conditions S and C are satisfied by all terms, then these rules, combined with the print_list program above, represent a reduction scheme for **Lists1** which is lazy. We can then use the PROLOG interpreter for actually carrying out the reductions, i.e. for "applying" reduction rules to terms in **Lists1**. By the completeness of PROLOG [13], if, for any term E, there exists an X such that reduce(E, X) is a consequence of the reduce rules, then PROLOG will find the value of X in finite time. It now remains to show:

Proposition. In the context of the reduction rules for **Lists1** *above, we have, for all terms E in* **Lists1**, $S(E)$ *and* $C(E)$.

PROOF. We use proof by structural induction, by considering each of the formation rules in **Lists1**.

[ ]. From the rule for [ ] we directly verify $S([\,])$ and $C([\,])$.

$[U|V]$ *where U is an irreducible element and V belongs to* **Lists1**. From rule for $[U|V]$ we also directly verify $S([U|V])$ and $C([U|V])$.

intfrom(N) *where N is an irreducible element and an integer.* For the condition S we see, in the context of the rule for intfrom(N), that if reduce(intfrom(N), Z) then Z is $[N|\text{intfrom}(N1)]$, where $N1$ is $N+1$. The value of intfrom(N) is clearly the value of $[N|\text{intfrom}(N1)]$, as required. For the condition C we know that the value of intfrom(N) is always a nonempty list. If we let $N1$ be the irreducible element equal to $N+1$ and V be intfrom($N1$), then we have, by the rule for intfrom(N), reduce(intfrom(N), $[N|\text{intfrom}(N1)]$). N is an irreducible element and intfrom($N1$) is a term, as required.

append(X, Y) *where X and Y belong to* **Lists1**. We assume that the conditions S and C hold for terms X and Y. We then show that these conditions hold for the term append(X, Y). For condition S we assume that the value of

append(X, Y) exists and that reduce(append(X, Y), Z) holds. Since this can only hold by one of the two rules for append(X, Y), we have one of two cases:

(1) reduce(X,[ ]) and reduce(Y, Z) hold (in the first rule). By hypothesis, the value of X is the empty list, and the value of Y is the value of Z. The value of Z is then the same as the value of append(X, Y), as required.

(2) reduce(X,[$FX|RX$]) holds (in the second rule) and Z is [FX|append(RX, Y)]. Using the hypothesis, we see that the value of Z is the same as the value of append(X, Y), as required.

For the condition C we once again have two cases:

(1) The value of append(X, Y) is the empty list. Then values of X and Y are also the empty list. Hence, by hypothesis, reduce(X,[ ]) and reduce(Y,[ ]), and so by the first rule, reduce(append(X, Y),[ ]), as required.

(2) The value of append(X, Y) is a nonempty list. If the value of X is a nonempty list, then, by hypothesis, we have reduce(X,[$FX|RX$]) for some irreducible element FX and term RX. By the second rule, reduce(append(X, Y),[FX|append(RX, Y)]), as required. If the value of X is the empty list, then the value of Y must be a nonempty list. Hence, by hypothesis we have reduce(X,[ ]) and reduce(Y,[$FY|RY$]) for some irreducible element FY and term RY. Then, by the second rule we have reduce(append(X, Y),[$FY|RY$]), as required.

filter(A, X) *where A is an integer and X belongs to* **Lists1.** The proof for this case is different from that for append, since the third rule contains the recursive call reduce(filter(A, RX), Z). We only outline the proof for the case when the value of filter(A, X) is an infinite list. There are two subcases. First, X does not contain an occurrence of filter. Then, on the basis of preceding steps in the proof, we know that every element of value of X can be obtained in finite time. We can now argue, by considering how PROLOG interprets reduce rules, that every element of filter(A, X) can also be obtained in finite time, in particular that S and C hold for filter(A, X). Second, if X does contain an occurrence of filter, then there must be a smallest subexpression of the form filter(U, V) such that V does not contain an occurrence of filter. Then every element of filter(U, V) can be obtained in finite time. We can now use structural induction to show that S and C also hold for filter(A, X).

Thus we can infer, via induction, the proposition. □

The practical consequence of this result is that if we wish to compute a list L which is the value of a term E in **Lists1**, we can reduce E using the reduction rules for **Lists1** and compute L lazily. In **Lists1** however we can only reduce applications of a fixed set of functions (i.e. [ ], |, intfrom, append, filter) to their arguments. To reduce applications of other functions we must extend **Lists1** by introducing, for each function, a formation rule and a set of reduction rules. We must then re-prove the proposition above. But we can do this by proving that the condition S and C are satisified for terms introduced by new formation rules, in the context of the extended set of reduction rules, and considering the proof as part of the induction proof above. Then, once again, we can be sure that terms in the extended language will be reduced lazily.

7. PREVENTING REPEATED EVALUATION OF EXPRESSIONS

We now show how we may preprocess programs and ensure, using logical variables, that values of expressions once generated are remembered for future access.

The following rule forms part of the definition of the function merge:

reduce(merge(U, V), [FU|merge(RU, V)]) :-

reduce(U, [FU|RU]), reduce(V, [FV|RV]), $FU < FV$.

Here U and V represent two sorted lists which have to be merged into a single sorted list. Their heads are obtained by the two successive calls to reduce on the right hand side. However, even if U and V happen to be the same expression, the head will be computed twice, once from each call to reduce, which could be expensive if the expression is a complex one. This is a problem similar to that in a "call by name" transmission of actual parameters to a procedure in ALGOL.

If U and V are the same, we would like V to have access to the value that U is reduced to. This could be done by attaching an extra output variable to each expression which would "hold on" to the value that the expression is reduced to. Different occurrences of an expression would then have the same output variable and so would simultaneously "feel" the binding of this variable to some value. When these occurrences are to be reduced, they first check if this variable is bound and if so, simply read off the value. Otherwise they are reduced using the definition of the appropriate function. It is important to stress that this extra variable is purely an optimization device and does not in any way affect the logic of the program. This is clear because values of output variables do not affect values of existing variables. So the logic of the program is unaffected.

However, since U and V are function applications, it is awkward to attach output variables to expressions in source code. Instead we have this done automatically by means of a preprocessor. Such a preprocessor has been written and the code is available from the author. To understand its operation we go through an example. The reduce rules for function append are

reduce(append(X, Y), Z) :- reduce(X, [ ]), reduce(Y, Z).

reduce(append(X, Y), [FX|append(RX, Y)]) :- reduce(X, [FX|RX]).

Attaching output variables to expressions of the form append(M, N), we obtain

reduce(append(X, Y, Result), Z) :- reduce(X, [ ]), reduce(Y, Z).

reduce(append(X, Y, Result), [FX|append(RX, Y, Output)]) :-

reduce(X, [FX|RX]).

Since the second argument of reduce represents the value of the first argument we need to connect the output variable of the first argument to the second argument. We finally obtain

reduce(append(X, Y, Z), Z) :- reduce(X, [ ]), reduce(Y, Z).

reduce(append(X, Y, [FX|append(RX, Y, Output)]),

[FX|append(RX, Y, Output)]) :-

reduce(X, [FX|RX]).

To ensure that an expression of the form append(X, Y, Z) is not reevaluated, we also have to add *before* these rules the assertion

reduce(append(X, Y, Z), $Z1$) :- not(var(Z)), !, $Z1 = Z$.

The first two arguments to append are always ground terms, but the third argument starts off as an unbound variable and accumulates members of the result as they are generated. Thus in the query

Q = append([1,2,3,4], [ ], Result), reduce(merge(Q, Q), Z).

the first call to reduce inside merge will bind Result to [1|append([2,3,4],[ ])] using the rules about append. The second call to reduce inside merge will first check, using the topmost assertion, if Result is bound. Since it is, the head of Q is read off as 1, and due to the presence of the cut (!) in the topmost rule, is not recomputed, i.e., the lower rules are not tried. It should be noted that this cut does not affect the nondeterminism as described in Section 4. For example, in the query reduce(perm([1,2], X), Z), X would initially be unbound, so both rules for perm would be tried. Thus, both permutations would be computed.

Of course, we do not have to preprocess all functions. In particular, we do not attach output variables to PROLOG lists, since they are already in irreducible form. This method of "updating" could easily be generalized to other data types.

8. TRANSLATING FUNCTIONAL LANGUAGE PROGRAMS

We now define a simple functional language and show how to translate programs in it into equivalent programs using our technique. That is, when a program correctly and completely defines functions in some language **Listsx** analogous to **Lists1**, its translation yields a set of reduce rules in whose context the conditions S and C are satisfied by all terms of **Listsx**. *Preparatory to the translation, the section presents a more operational explanation of our technique.*

A functional language program consists of a collection of reduction rules, each of the form:

LHS $\Rightarrow$ RHS :- $Q1..Qn$. $\quad n >= 0$.

where LHS and RHS are l-expressions and each Qi is a PROLOG condition. An l-expression is a function application of the form $f(t1, \ldots, tk)$, $k >= 0$, where f is a k-ary function symbol in **Listsx** and each ti is either a PROLOG variable, a constant in **Listsx**, or an l-expression. *In* LHS *only, when ti denotes a list, we further restrict ti to be an l-form.* An l-form is either [ ], a variable, or $[U|V]$, where U is either an irreducible element or a variable and V is an l-form. We place the additional restriction that none of the Qi be a call to $\Rightarrow$. These restrictions are primarily to keep simple the translation procedure described below, and do not cause much loss of expressive power.

The reading of such a rule is that a term in **Listsx** which unifies with LHS with m.g.u. s reduces to RHS(s) provided each of condition $Qi(s)$ is satisfied, where $E(s)$ denotes the result of applying substitution s to expression E. For example, we

can describe functions intfrom and filter as follows:

intfrom(N) $\Rightarrow$ [N|intfrom($N1$)] :- $N1$ is $N+1$.

filter(A, [ ]) $\Rightarrow$ [ ].

filter(A, [U|V]) $\Rightarrow$ [U|filter(A, V)] :- not multiple(U, A).

filter(A, [U|V]) $\Rightarrow$ filter(A, V) :- multiple(U, A).

$\Rightarrow$ rules are clearer and more concise than the equivalent reduce rules. However, we cannot use them directly; e.g., the query filter(2, intfrom(4)) $\Rightarrow Z$ will fail. We need to define a reduction strategy for reducing an expression E into list form, whenever the value of E is a list. By list form we mean [ ] or [U|V] where U is an irreducible element and V is a term in **Listsx**.

The strategy is based upon answers to two questions, both directly relevant to lazy evaluation. First, if we wish to reduce $f(t1, \ldots, tn)$ by the rule $f(s1, \ldots, sn) \Rightarrow E2$:- Q, to what extent must we attempt to reduce ti (when ti denotes a list) before we try to unify it with si? Second, when do we terminate a sequence of applications of $\Rightarrow$ rules?

The first question can be answered easily by noting that we must attempt to reduce ti till it acquires the form of si. For example, if we wish to reduce filter(2, intfrom(2)) by the third filter rule, we must reduce intfrom(2) to the form [M|N] before we try to unify filter(2, [M|N]) with filter (A, [U|V]). Information about the form of si is, of course, available at compile time, this being the reason for restricting si to be an l-form. Therefore it can be explicitly stored and used by the reduction strategy at run time. Since it does not have to be inferred at run time, the reduction is considerably speeded up.

The answer to the second question is simply that reduction terminates as soon as it produces an expression in list form. Suppose $f(t1, \ldots, tn)$ unifies with $f(s1, \ldots, sn)$ with m.g.u. d. If $E2(d)$ is in list form and $Q(d)$ is satisfied, then reduction can terminate. If $E2(d)$ is not in list form, we recursively apply $\Rightarrow$ rules to $E2(d)$.

Based upon the above considerations, we now implement a reduction strategy using the predicate simplify. Given an expression E which denotes a list, if simplify(E, Z) succeeds, then Z is in list form and the value of Z is the same as the value of E:

simplify(X, Y) :- $X \Rightarrow Y$, list_form(Y).

simplify(X, Y) :- $X \Rightarrow Z$, not(list_form(Z)), simplify(Z, Y).

simplify(X, Y) :- preprocess(X, Z), not($X = Z$), simplify(Z, Y).

list_form([ ]).

list_form([U|V]).

For every $\Rightarrow$ rule for the function f, preprocess defines the extent to which arguments of f must be reduced in order for that rule to be used for reducing $f(t1, \ldots, tn)$. Preprocess clauses can be derived by a simple compile time analysis of

$\Rightarrow$ rules. For example, we have for filter the clauses

preprocess(filter(A, Q), filter(A, [ ])) :- simplify(Q, [ ]).

preprocess(filter(A, Q), filter(A, [B|C])) :- simplify(Q, [B|C]).

Now simplify(fiter(2, intfrom(2)), Z) will halt with Z = [3|intfrom(4)].

We now raise the possibility of combining the logic of $\Rightarrow$ rules and the strategy in simplify and preprocess rules into a single specification. It is the realization of this possibility, in reduce rules, which forms the basis of our approach to lazy evaluation. The combined specification not only has a natural, logical reading (Section 6) but is also efficient.

We now describe a procedure for deriving reduce rules from $\Rightarrow$ rules. To formally prove its correctness we need to show that for all E denoting a list, simplify(E, $E1$) iff reduce(E, $E1$). We do not show this here. However, some informal justification can be obtained by noting that the main steps are (b) and (c). Step (b) derives information from preprocess rules, and step (c) implements the decision to terminate or to continue.

Any new variable introduced below is different from those occurring in the rule being translated, and different from those introduced at earlier stages. The rules [U|V] $\Rightarrow$ [U|V] and [ ] $\Rightarrow$ [ ] are translated respectively into

reduce([U|V], [U|V]).

reduce([ ], [ ]).

The translation of LHS $\Rightarrow$ RHS :- $Q1, \ldots, Qn$. $n >= 0$ proceeds in five steps:

(a) LHS. LHS is of the form $f(X1, \ldots, Xn)$. Set the variable LHS* to $f(A1, \ldots, An)$, where each Ai is a PROLOG variable.

(b) *Argument Xi in* LHS. If Xi is [ ], generate the condition reduce(Ai, [ ]). If Xi is of the form [$U1, U2 \ldots Un$|V], generate the conditions reduce(Ai, [$U1$|$W1$]), reduce($W1$, [$U2$|$W2$]), ..., reduce(Wn-1, [Un|V]), where Wi are new variables. Otherwise, set Ai to Xi.

(c) RHS. If RHS is of the form [ ] or [U|V], set the variable RHS* to RHS. Otherwise, set the variable RHS* to the variable Out and generate the condition reduce(RHS, Out).

(d) $Q1, \ldots, Qn$. These are left intact.

(e) *Full rule.* Finally generate the rule

reduce(LHS*, RHS*) :- LHS_Conds + ($Q1, \ldots, Qn$) + RHS_Conds.

where + concatenates sets of conditions, and LHS_Conds and RHS_Conds are sets of conditions generated in translating respectively LHS and RHS.

9. CONNECTIONS WITH PREVIOUS WORK

9.1. Complete Procedures and Lazy Evaluation

The ideas in this paper began to develop upon consideration of Turner's [24] observation that a normal-order reduction strategy for reducing lambda expressions

directly yields lazy evaluation. This strategy leads to the normal form of an expression whenever one exists, and so may be called "complete". By analogy then, we may be able to get lazy evaluation in logic by the use of complete proof procedures such as resolution for the first-order predicate calculus or LUSH resolution [13] for Horn clauses. We obtain further motivation in this direction by the statement of the compactness theorem of first-order predicate calculus [21]: If an infinite set of clauses is unsatisfiable, then it has a finite subset which is also unsatisfiable. But a complete proof procedure could find this set in finite time. As with normal-order reduction, we could again get an "answer" in finite time even with an "infinity" in the input. So, if we could represent data structures as sets of clauses, complete proof procedures could also give us lazy evaluation directly.

Such considerations led to a closer look at Kowalski's [17] ideas regarding representing data as relations. If x is the list $[a, b]$, then Kowalski suggests defining x by the following set of assertions:

item$(x, 1, a)$.

item$(x, 2, b)$.

where item(x, N, E) means that the Nth element of the list x is E. To define a nonempty list, instead of specifying each element of that list, it is enough to specify just its head and tail. For example, the list intfrom(N) can be defined using the following two assertions:

head(intfrom(N), N).

tail(intfrom(N), intfrom$(N1)$) :- $N1$ is $N + 1$.

These two assertions can be combined into

reduce(intfrom(N), $[N|$intfrom$(N1)]$) :- $N1$ is $N + 1$.

Extending this idea to other functions led to our technique for lazy evaluation and to identification of lazy evaluation with completeness (Section 6).

9.2. *Functional Programming*

The above approach for defining functions can also be followed in functional programming: if an expression denotes a list, define what its head and tail are. For example, the expression intfrom(N) can be defined as a function:

$$\text{intfrom}(N) = \lambda(\text{sel})\ \text{if eq}(\text{sel}, \text{head})\ \text{then } N\ \text{else } [\text{if eq}(\text{sel}, \text{tail})\ \text{then intfrom}(N+1)]$$

Note that the traditional roles of data structure and function application have been reversed. The function intfrom(N) only accepts head and tail as arguments. The expression [[intfrom(1)](tail)](head) would yield 2 in finite time. In a LISP such as the *T* dialect [20] which handles lambda expressions properly, we can use the above approach for defining functions to get lazy evaluation without change to the LISP interpreter.

Lazy evaluation is, in fact, already inherent in the LISP interpreter. A function f which is defined recursively represents an infinite structure. To obtain it, repeatedly

substitute the definition of f in the definition of f. However, when LISP is given such a definition, it does not immediately proceed to expand it in such a way. Such an attempt would never terminate. Rather, LISP waits till the argument is specified and then, depending on it, determines the extent to which the definition is to be expanded. Thus the argument represents "demand". So, if we can represent data structures as functions and selector functions as arguments, then data structures would expand only upon demand, i.e. lazily. This situation is realized by the method of defining intfrom above, and similarly can easily be realized for other functions.

9.3. *Concurrent* PROLOG

Since our technique suggests a practical method of dealing with infinite data structures, it is quite feasible to implement functional versions of communicating processes, e.g. functional operating systems [12]. However, we are not restricted to a purely functional style. We give a program in Shapiro's Concurrent PROLOG [22] and then rewrite it using our technique to show how we can simulate its behavior, in particular that of read-only variables. A FIFO queue process in [22] is defined as

queue([enqueue(X)|L], Head, [X|Tail]) :- queue(L?, Head, Tail).

queue([dequeue(X)|L], [X|Head], Tail) :- queue(L?, Head, Tail).

The first argument is a list of commands either for adding an element to the end of the queue or for deleting an element from the front of the queue. Given an input stream of commands S, the process is started with the query

queue(S, H, H).

A read-only variable such as L? ensures that the queue process on the right-hand side waits till L? is bound to a list, i.e., it waits till S produces the next command.

If S is represented by a functional expression, then we can rewrite the above program without the use of read-only variables:

queue(S, [ ], [ ]) :- reduce(S, [ ]).

queue(S, [X|Head], Tail) :- reduce(S, [dequeue(X)|RS]), queue(RS, Head, Tail).

queue(S, [X|Head], Tail) :- reduce(S, [dequeue(X)|RS]), queue(RS, Head, Tail).

In general, when S is a function application, RS will also be one. The recursive call to queue will lead to a call to reduce, which will then wait till RS can be reduced. For example, S could be some function produce defined as

reduce(produce, [enqueue(2), dequeue(X), enqueue(3), dequeue(Y)]).

The process, when started by the query

queue(produce, H, H).

will insert and delete elements in the queue just as in Concurrent PROLOG.

A major use of a read-only variable is representing an uncomputed portion of a stream. Processes which contain this variable must wait till this portion is computed further. We can also represent a portion of a stream by a function application. A process containing this function application will not proceed till it can be reduced. We can thus simulate this use of read-only variables. Of course, to do true concurrent processing we still require the presence of at least OR-parallelism.

We can thus simulate this use of read-only variables. Of course, to do true concurrent processing we still require the presence of at least OR-parallelism.

10. SUMMARY AND DIRECTIONS FOR FUTURE WORK

We have presented a natural and efficient technique for defining functions in logic that directly yields lazy evaluation.

The technique shows how we can achieve lazy evaluation by keeping the interpreter fixed but modifying the programming style. However, the style continues to be natural. Since programs written using it run as PROLOG programs, they run very efficiently. This is in contrast to previous approaches for achieving lazy evaluation, which have advocated keeping the programming style fixed but modifying the interpreter. However, we also show how to translate programs in a simple functional language into programs using the new technique.

The technique, moreover, does not restrict one to a purely functional style. It can be used freely in combination with usual PROLOG programs, and so one continues to benefit fully from the powerful features of PROLOG, principally, unification, nondeterminism, partially instantiated structures, and a deductive apparatus.

A simple preprocessing technique, utilizing the nature of logical variables, ensures that repeated evaluation of the same expression does not take place. We do not need to explicitly maintain environments for variables, since they are automatically maintained for us by PROLOG. Secondary indexing of clauses would further improve the efficiency of programs.

Directions we now plan to investigate include: (1) formalization within the framework of Section 6, of lazy evaluation for trees and other data structures, of nondeterminism, and of the idea of considering relations as functions (e.g. safe in Section 4); (2) applications of lazy evaluation e.g. to process representation, object oriented programming, or parsing.

I would like to thank Professor Stott Parker for very helpful discussions on this paper. I would also like to thank Professor J. A. Robinson and the referees for their very helpful reviews. Finally, I thank Philip Klahr, Randall Steeb, David McArthur, Stephanie Cammarata, and Jed Marti for their criticisms and suggestions. David McArthur's assistance in formatting this paper is also greatly appreciated.

Note added in proof: In the second paragraph of section 8.0 we place the following additional restrictions on $\Rightarrow$ rules: (a) Suppose tj is $[W1,\ldots,Wk|Vk]$ where Vk is a variable. If ti is a variable V, V is different from Vk. If ti is $[U1,\ldots,Un|Vn]$ where Vn is a variable, Vn is also different from Vk. (b) If ti is a variable V, then V does not appear in any of the Qi. If ti is $[U1,\ldots,Un|Vn]$, where Vn is a variable, Vn also does not appear in any of the Qi.

REFERENCES

1. Barendregt, H. P., The Type Free Lambda Calculus, in: John Barwise (ed.), *Handbook of Mathematical Logic*, North Holland, 1977.
2. Burstall, R. and Darlington, J., A Transformation System for Developing Recursive Programs, *J. Assoc. Comput. Mach.* 24, No. 1 (1977).
3. Church, A., *The Calculi of Lambda Conversions*, Annals of Mathematical Studies, No. 6, Princeton, U.P., Princeton, 1941.

4. Clark, K. L., Predicate Logic as a Computational Formalism, Research Monograph, Imperial College, Univ. of London, 1980.
5. Curry, H. B. and Feys, R., *Combinatory Logic*, Vol. I, North Holland, Amsterdam, 1958.
6. Friedman, D. P. and Wise, D. S., CONS should not evaluate its arguments, in: S. Michaelson, R. Milner (eds.), *Automata, Languages and Programming*, Edinburgh U.P., Edinburgh, 1976.
7. Gallagher, J., Simulating Coroutining for the 8 Queens Problem, *Logic Programming Newsletter* 3 (Summer 1982).
8. Gallaire, H. and Lasserre, C., Metalevel Control for Logic Programs, in: K. Clark and S. A. Tarnlund (eds.), *Logic Programming*, Academic, New York, 1982.
9. Guttag, J. V., Abstract Data Types and the Development of Data Structures, *Comm. ACM* 20, No. 6 (1977).
10. Hansson, A., Haridi, S., and Tarnlund, S. A., Properties of a Logic Programming Language, in: K. Clark and S. A. Tarnlund, (eds.), *Logic Programming*, Academic, New York, 1982.
11. Henderson, P., *Functional Programming: Application and Implementation*. Prentice-Hall, Englewood Cliffs, N.J., 1980.
12. Henderson, P., Purely Functional Operating Systems, in: J. Darlington, P. Henderson, and D. A. Turner (eds.), *Functional Programming and its Applications, An Advanced Course*, Cambridge U.P., New York, 1982.
13. Hill, R., LUSH Resolution and Its Completeness, DCL Memo 78, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1974.
14. Kahn, K., Actors in Prolog, in: K. Clark and S. A. Tarnlund (eds.), *Logic Programming*, Academic, New York, 1982.
15. Kahn, G. and McQueen, D., Coroutines and Networks of Parallel Processes, in: *Information Processing—77*, North-Holland, Amsterdam, 1977.
16. Kornfeld, W., Equality for Prolog, in: *Proceedings of IJCAI-83*, Karlsruhe, West Germany, 1983.
17. Kowalski, R., *Logic for Problem Solving*, Elsevier North Holland, New York, 1979.
18. Liskov, B. and Zilles, S. Programming with Abstract Data Types, in: *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, SIGPLAN Notices* 9:4 (1974).
19. Narain, S., A Technique for Doing Lazy Evaluation in Logic, in: *Proceedings of the Second IEEE International Symposium on Logic Programming*, Boston, 1985.
20. Rees, J., Adams, N., and Meehan, J., *The T Manual*, Computer Science Dept., Yale Univ., New Haven, 1983.
21. Robinson, J. A., *Logic: Form and Function. The Mechanization of Deductive Reasoning*, Elseiver North Holland, New York, 1979.
22. Shapiro, E. and Takeuchi, A., Object-Oriented Programming in Concurrent Prolog, in: *New Generation Computing I (1983)*, OHMSA and Springer, Japan, 1983.
23. Subrahmanyam, P. A. and You, J.-H., Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming, in: *Proceedings of 1984 IEEE Logic Programming Symposium*, Atlantic City, N.J., 1984.
24. Turner, D., A New Implementation Technique for Applicative Languages, *Software Practice and Experience*, 9:31–49 (1979).